MAPSI

Modèles et Algorithmes de Probabilités et Statistiques pour l'Informatique

TME: Chaine de Markov: classification de lettres manuscrites

Téléchargement des données

Nous travaillerons sur des lettres manuscrites:

TME6_lettres.pkl

Classification de lettres manuscrites

Format des données

Les données sont fournies au format pickle (le standard de sérialisation python, particulièrement convivial). Pour les charger:

```
import numpy as np
import pickle as pkl
import matplotlib.pyplot as plt

data = pkl.load(file("ressources/lettres.pkl","rb"))
X = np.array(data.get('letters')) # récupération des données sur les lettres
Y = np.array(data.get('labels')) # récupération des étiquettes associées
```

[$[Get Code]]

Les données sont dans un format original: une lettre est en fait une série d'angles (exprimés en degrés). Un exemple:

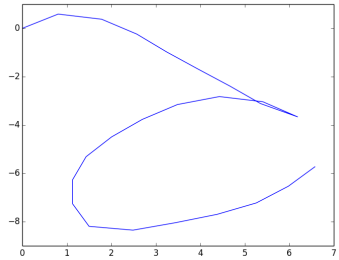
```
X[0] = array([  36.214493,  347.719116,  322.088898,  312.230957,  314.851013,
        315.487213,  313.556702,  326.534973,  141.288971,  167.606689,
        199.321594,  217.911087,  226.443298,  235.002472,  252.354492,
        270.045654,  291.665161,  350.934723,   17.892815,   20.281025,
         28.207161,   43.883423,   53.459026])
```

Lors de l'acquisition, un stylo intelligent a pris des mesures régulièrement dans le temps: chaque période correspond à un segment de droite et le stylo a calculé l'angle entre deux segments consécutifs... C'est l'information qui vous est fournie.

Pour afficher une lettre, il faut reconstruire la trajectoire enregistrée... C'est ce que fait la méthode ci-dessous:

```
# affichage d'une lettre
def tracerLettre(let):
    a = -let*np.pi/180; # conversion en rad
    coord = np.array([[0, 0]]); # point initial
    for i in range(len(a)):
        x = np.array([[1, 0]]);
        rot = np.array([[np.cos(a[i]), -np.sin(a[i])],[ np.sin(a[i]),np.cos(a[i])]])
        xr = x.dot(rot) # application de la rotation
        coord = np.vstack((coord,xr+coord[-1,:]))
    plt.figure()
    plt.plot(coord[:,0],coord[:,1])
    plt.savefig("exlettre.png")
    return
```

[$[Get Code]]



Apprentissage d'un modèle CM (max de vraisemblance)

1. Discrétisation

1 état = 1 angle

Il est nécessaire de regrouper les angles en un nombre fini d'états (par exemple 20)

- définir un $intervalle = 360 . / n_{etats}$
- discrétiser tous les signaux x à l'aide de la formule

```
np.floor(x/intervalle)
```

Donner le code de la méthode `discretise(X,d)` qui prend la base des signaux et retourne une base de signaux discrétisés.

VALIDATION code du premier signal avec une discrétisation sur 3 états:

SEARCH
Go

TUTORIEL NUMPY

INFOS COURS/TD/TME

SEMAINIER

Semaine 1
Semaine 2
Semaine 3
Semaine 4
Semaine 5
Rapport 1
Semaine 6
Semaine 7?
Semaine 8
Semaine 9?
Semaine 10?

LIENS

edit SideBar

```
array([ 0.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  1.,  1.,  1.,  1.,  1.,
        1.,  2.,  2.,  2.,  2.,  0.,  0.,  0.,  0.,  0.])
```

2. Regrouper les indices des signaux par classe (pour faciliter l'apprentissage)

```
def groupByLabel( y):
    index = []
    for i in np.unique(y): # pour toutes les classes
        ind, = np.where(y==i)
        index.append(ind)
    return index
```

[$[Get Code]]

Cette méthode produit simplement une structure type:

```
[array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]),
array([22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]),
array([33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43]),
array([44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54]),
array([55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65]),
...
```

Chaque ligne regroupe les indices de signaux correspondant à une classe

3. Apprendre les modèles CM

Soit X_C la base de signaux discrétisés correspondant à une classe C et d le nombre d'états. Donner le code de la méthode `def learnMarkovModel(Xc, d)` qui retourne un tuple contenant Pi et A.

Rappel:

- Initialisation de

```
A = np.zeros((d,d))
Pi = np.zeros(d)
```

[$[Get Code]]

- Parcours de tous les signaux et incréments de A et Pi
- Normalisation (un peu réfléchie pour éviter les divisions par 0)

```
A = A/np.maximum(A.sum(1).reshape(d,1),1) # normalisation
Pi = Pi/Pi.sum()
```

[$[Get Code]]

VALIDATION premier modèle avec une discrétisation sur 3 états:

```
(array([ 0.36363636,  0.        ,  0.63636364]),
array([[ 0.84444444,  0.06666667,  0.08888889],
       [ 0.        ,  0.83333333,  0.16666667],
       [ 0.11382114,  0.06504065,  0.82113821]]))
```

4. Stocker les modèles dans une liste

Pour un usage ultérieur plus facile, on utilise le code suivant:

```
d=20     # paramètre de discrétisation
Xd = discretise(X,d)  # application de la discrétisation
index = groupByLabel(Y)  # groupement des signaux par classe
models = []
for cl in range(len(np.unique(Y))): # parcours de toutes les classes et optimisation des
modèles
    models.append(learnMarkovModel(Xd[index[cl]], d))
```

[$[Get Code]]

Test (affectation dans les classes sur critère MV)

1. (log)Probabilité d'une séquence dans un modèle

Donner le code de la méthode `probaSequence(s,Pi,A)` qui retourne la log-probabilité d'une séquence s dans le modèle $\lambda = \{Pi, A\}$

VALIDATION probabilité du premier signal dans les 26 modèles avec une discrétisation sur 3 états:

```
array([-13.491086  ,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf, -12.48285678])
```

- Ce signal est-il bien classé?
- D'où viennent tous les `-inf`?

2. Application de la méthode précédente pour tous les signaux et tous les modèles de lettres

L'application se fait en une ligne de code si vous avez respecté les spécifications précédentes:

```
proba = np.array([[probaSeq(Xd[i], models[cl][0], models[cl][1]) for i in range(len(Xd))]for
cl in range(len(np.unique(Y)))])
```

[$[Get Code]]

3. Evaluation des performances

Pour l'évaluation, nous proposons l'approche suivante:

- calcul d'une version numérique des Y

```
Ynum = np.zeros(Y.shape)
for num,char in enumerate(np.unique(Y)):
    Ynum[Y==char] = num
```

[$[Get Code]]

- Calcul de la classe la plus probable:

```
pred = proba.argmax(0) # max colonne par colonne
```

[$[Get Code]]

- Calcul d'un pourcentage de bonne classification

```
print np.where(pred != Ynum, 0.,1.).mean()
```

[$[Get Code]]

INDICE DE PERFORMANCE : 91% de bonne classification avec 20 états, 69% avec 3 états

Biais d'évaluation, notion de sur-apprentissage

Dans le protocole précédent, nous avons triché:

- les données servent d'abord à apprendre les modèles...
- puis nous nous servons des mêmes données pour tester les modèles! Les performances sont forcément bonnes!

Afin de palier le problème, nous allons diviser en deux la base de données: une partie servira à l'apprentissage des modèles, l'autre à leur évaluation. Pour effectuer la division, nous fournissons le code suivant:

```
# separation app/test, pc=ratio de points en apprentissage
def separeTrainTest(y, pc):
    indTrain = []
    indTest = []
    for i in np.unique(y): # pour toutes les classes
        ind, = np.where(y==i)
        n = len(ind)
        indTrain.append(ind[np.random.permutation(n)][:np.floor(pc*n)])
        indTest.append(np.setdiff1d(ind, indTrain[-1]))
    return indTrain, indTest
# exemple d'utilisation
itrain,itest = separeTrainTest(Y,0.8)
```

[$[Get Code]]

dans `itrain`, nous obtenons les indices des signaux qui doivent servir en apprentissage pour chaque classe:

```
itrain = [array([ 6,  1,  2,  9, 10,  0,  4,  3]), # 80% des signaux de
array([16, 19, 20, 12, 21, 11, 15, 18]), # la classe 0 (='a') + permutation aléatoire
array([27, 25, 28, 26, 31, 29, 32, 23]),
array([42, 38, 34, 36, 41, 35, 33, 40]),
array([44, 48, 53, 50, 46, 45, 54, 49]),
array([63, 55, 58, 64, 56, 65, 60, 62]),
array([66, 74, 76, 67, 71, 73, 68, 72]),
```

Note: pour faciliter l'évaluation des modèles, vous aurez besoin de re-fusionner tous les indices d'apprentissage et de test. Cela se fait avec les lignes de code suivantes:

```
ia = []
for i in itrain:
    ia += i.tolist()
it = []
for i in itest:
    it += i.tolist()
```

[$[Get Code]]

Questions importantes

- Ré-utiliser les fonctions précédemment définies pour apprendre des modèles et les évaluer sans biais.
- Donner les résultats obtenus en apprentissage et en test
- Etudier l'évolution des performances en fonction de la discrétisation

Evaluation qualitative

Nous nous demandons maintenant où se trouvent les erreurs que nous avons commises...

Calcul de la matrice de confusion: pour chaque échantillon de test, nous avons une prédiction (issue du modèle) et une vérité terrain (la vraie étiquette). La matrice de confusion est une matrice (Nc x Nc) où nous comptons le nombre d'échantillon de test dans chaque catégorie:

- Initialisation à 0:

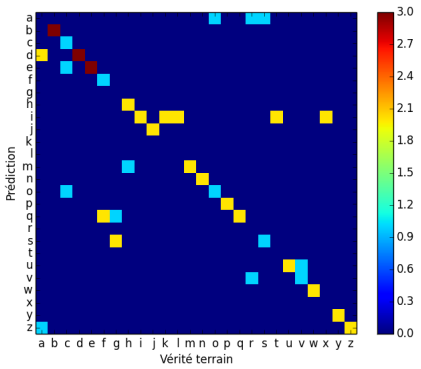
```
conf = np.zeros((26,26))
```

[$[Get Code]]

- Pour chaque échantillon, incrément de la case (prediction,vérité)
- Tracé de la matrice:

```
plt.figure()
plt.imshow(conf, interpolation='nearest')
plt.colorbar()
plt.xticks(np.arange(26),np.unique(Y))
plt.yticks(np.arange(26),np.unique(Y))
plt.xlabel(u'Vérité terrain')
plt.ylabel(u'Prédiction')
plt.savefig("mat_conf_lettres.png")
```

[$[Get Code]]



Dans l'exemple ci-dessus, nous voyons par exemple que:

- tous les exemples de 'b' sont bien reconnus
- tous les exemples de 'a' sont mal classés (un dans 'z', deux dans 'd')

Modèle génératif

Utiliser les modèles appris pour générer de nouvelles lettres manuscrites.

Tirage selon une loi de probabilité discrète

- faire la somme cumulée de la loi sc
- tirer un nombre aléatoire t entre 0 et 1
- trouver la première valeur de sc qui est supérieure à t
- retourner cet état

Génération d'une séquence de longueur N

- tirer un état s_0 selon Pi
- tant que la longueur n'est pas atteinte:
  - tirer un état s_{t+1} selon $A[s_t]$

Affichage du résultat

```
newa = generate(models[0][0],models[0][1], 25) # generation d'une séquence d'états
intervalle = 360./d # pour passer des états => valeur d'angles
newa_continu = np.array([i*intervalle for i in newa]) # conv int => double
tracerLettre(newa_continu)
```

[$[Get Code]]